# Chapter 1

# Towards DVM Friendly First-Class Functions

Marco T. Morazán[1]

***Abstract:*** One of the most attractive features of functional languages is that functions are first-class. To support first-class functions many functional languages created heap-allocated closures to store the bindings of free variables. This makes it difficult to predict how the heap is accessed and makes accesses to free variables slower than accesses to bound variables at runtime. This article presents how support for first-class functions is provided in the MT Evaluator Virtual Machine without creating heap-allocated closures by using partial evaluation at runtime.

## 1.1 INTRODUCTION

One of the most attractive features of functional languages is that functions are first-class. That is, functions can be passed in as arguments to functions and functions can be returned as values from functions. First-class functions provide programmers with powerful means of abstraction to capture computation patterns that help reduce development time and that result in elegant and easy to maintain code.

In order to support first-class functions, functional languages can not rely solely on runtime stack allocation of activation records to store variable bindings and flow control information. Consider the Scheme definition for $f$:

```
(define (f x) (lambda (y) (+ x y)))
```

that returns a function that adds $x$ to its input. In the scope of the returned function $x$ is a free variable. If the binding for $x$ when $f$ is evaluated were solely

[1]Department of Mathematics and Computer Science, Seton Hall University, South Orange, NJ, 07079; Phone: +1 973-761-9466; Fax: +1 973-275-2366; Email: morazanm@shu.edu; This work was partially funded by the University Research Council of Seton Hall University.

stored on the stack, it may be overwritten before the the function returned by $f$ is ever applied to an argument. To guarantee that the bindings of free variables are not lost, many functional languages create closures. A closure is a data structure that contains a pointer to a function and the bindings of the free variables in the function.

To implement closures, activation records can be heap-allocated allowing closures to point to those that hold the bindings of free variables[Chr90, AM91]. In this manner, even if an activation record is popped of the stack the bindings of free variable are not overwritten. Another approach is to have closures only store the bindings of free variables and use the stack to access bound variables[Luc83, Car84]. In both cases, closures are allocated in the heap creating the potential for poor locality of reference, for the reduction of intra-list locality[2], and for slow access to free variables by forcing accesses to the heap. The use of a generational garbage collector[Moo84, Wil92] can improve locality of reference in languages with closures[AS96], but does not address having to access the heap to resolve the value of free variables. Garbage collectors, including the generational flavor, interfere with program evaluation which is to some extent undesirable. To reduce the live data size and make accesses to free variables faster, safely linked closures[SA94] can be used which still, nonetheless, employ lists to implement closures and depend on garbage collection to guarantee locality of reference.

In this article, we present an implementation strategy for first-class functions that does not require the creation of heap-allocated closures. The technique is based on partial evaluation and on the generation of new functions at runtime for a pure functional language. Implementation support is described by adding primitive operations to the MT Evaluator Virtual Machine (MTEVM), whose semantics is given operationally by state transitions. First, relevant literature is reviewed to provide a common context to all readers. This is followed by a brief description of the MT system and results obtained from previous studies. The MTEVM itself is first described informally and then its operational semantics without support for first-class functions is given. This is followed by the introduction of new primitives needed to support first-class functions along with their operational semantics and hints on how an MT-Scheme compiler (for a pure subset of Scheme) can exploit these new primitives. Finally, conclusions, expectations, and new avenues of research created by this technique are presented.

## 1.2 RELATED WORK

### 1.2.1 Virtual Machines

Virtual machines (or abstract machines) permit the evaluation of programs, but omit details of real machines implemented in hardware. They are used to bridge the gap between high-level languages and hardware machines by tailoring their set

[2]Intra-list locality refers to the property that the elements of a list are allocated close to each other in the heap.

of primitive operations to specific operations needed by a class of languages[Ste00]. In addition, virtual machines are used to implement compilers, to increase maintainability and portability, and to facilitate the correctness of code generators. Some virtual machines have been implemented and are commercially used like the Java Virtual Machine[LY96].

The use of virtual machines has been popular in the implementation of functional languages. One of the earliest virtual machines is the SECD machine, first conceived by Landin[P. 64], which Henderson used for an implementation of Lisp[Hen80]. The SECD machine consists of only four registers. Henderson used transition rules to describe the semantics of the machine operationally. A similar approach was taken in Cardelli's definition of the Functional Abstract Machine (FAM) created to support functional languages, fast function application, and the use of real stacks[Luc83, Car84]. FAM is more complex than SECD consisting of six pointers, three stacks, and a data heap. Both machines define computation primitives that yield new values and flow control primitives that determine what part of the program is to be executed.

Most virtual machines for functional languages, in general, have a set of registers, one or more stacks, a data heap, a control, and support for environments and closures as the SECD and FAM machines. They may also include support for such things as interrupts like in the Scheme48 virtual machine[Joh02] or novel language features like, for example, regions for ML[Mad98] and parallel/distributed computing with lazy languages[S. 98, BF00, Loi01]. A survey of virtual machines for functional and other languages has been done by Diehl et. al.[Ste00].

### 1.2.2 Closure Representation

There have been several techniques used to represent closures in functional languages. In Henderson's SECD, a closure is a pair containing a list representing the control structure and a list of frames representing the environment. This is called a *linked closure*. To access the value of a variable its frame number and its displacement within the frame must be known and a traversal of the environment must ensue. In FAM, a closure is represented by a *closure cell* that contains (a pointer to) the text to be evaluated and the bindings of the free variables in the text. This is called a *flat closure*. In flat closures, free variables can be accessed by some fixed displacement within a closure cell, but many values may have be copied repeatedly from closure to closure as pointed out by Shao and Appel[SA94]. To avoid this copying, they developed *safely linked closures* that allow the sharing of bindings with the same lifetime. Safely linked closures, like flat closures and unlike linked closures, guarantee that a binding is unreachable after it is no longer relevant to the computation.

The closure representations described above require some type of memory allocation beyond allocation on a stack and make the access to free variables slower than the access to bound variables residing on the stack. The latter is unfortunate, because the values of free variables are known values (just as the values of bound variables) when the function represented by the closure is applied. Pushing

the values onto the stack before function application is not an efficient solution, because the closure must be traversed and each value accessed. Improving the access time to free variables has been a motivating factor in the work presented here.

### 1.2.3 Partial Evaluation

The roots of modern partial evaluation techniques can probably be traced back to lambda-conversion defined by Church[Chu41]. The value of partial evaluation in program evaluation (also known as program specialization) was realized by Dijkstra when he defined substitution processes that returned expressions to be evaluated at a future time[Dij62]. A partial evaluator is given a program, $P(x_1 \ldots x_n)$, along with some of $P$'s input, $x_1 \ldots x_i$, and builds a new program, $P_{new}$, which when it receives the rest of the input, $x_{i+1} \ldots x_n$, will return the same result as $P$ when given all of its input. $P_{new}$ can be the result of simply replacing the known values, $x_1 \ldots x_i$, in the body of $P$ or can be specialized to the point of executing operations in $P$ (including partial evaluation of function calls) based on the known values $x_1 \ldots x_i$. A partial evaluator, therefore, performs a mixture of code generation and execution[Jon96]. Special care, however, must be taken in the presence of side-effects, because referential transparency is not guranteed[Ken97].

Partial evaluation in functional languages has been used, for example, to implement runtime code generation for a pure subset of ML that does not include higher-order funtions[Pet96, Mar94]. The technique used goes beyond simply substituting values in a sequence of instructions for $P$. Instead, specialized code generators are created that do not need to process the original sequence of instructions when code is generated dynamically. The results reported lead to the conclusion that the technology is usable, but needs to be improved.

## 1.3 THE MT SYSTEM

An MT node is a parallel machine with one processing element dedicated to the evaluation of programs and the rest of the processing elements dedicated to memory management. The MT system divides memory into 5 distinct spaces that are managed independently and are implemented as 5 different address spaces in a distributed virtual memory (DVM) system implemented in software. These memory spaces are: the MT heap, the MT stack, the MT code space, the evaluator space, and the garbage collector space. Of these, the garbage collector space has not yet been designed or implemented. It would not be prudent to design a garbage collecting system for an MT node before having a thorough understanding of how MT spaces are accessed. Any garbage collector must enhance the performance of the evaluator and have knowledge of how live-data in the different memory spaces should be stored.

The virtual address spaces for the heap, the stack, and the code are each distributed and divided into pages. The evaluator allocates to each of these spaces exclusive use of a set of frames. A demand paging algorithm is implemented for

each space to swap pages between the evaluator's frames and each of the distributed MT memory spaces. In the current implementation of MT, the number of frames assigned to each memory space is fixed at runtime. This implementation choice, however, can be changed, as experienced is gained, to adjust dynamically the number of frames allocated for use to each MT space.

The heap and the stack operate on units of data called MT S-expressions. Each S-expression has a tag and a value. Tags distinguish the type of value that is held by the MT S-expression. Primitive types are represented by an integer. For example, an INT tag means that the value is an integer, a SYMB tag means that the value is an index into the symbol table, a FUNCT tag means that the value is an index into the function table, a PRIM tag means that the value is a primitive function (e.g. +, *, and cons) and a BOOL tag means that the value is a boolean. Compound types are represented by two integers that are indexes (i.e. addresses) into heap space. The only compound type is a list that is represented with a LIST tag and two addresses for the *car* and the *cdr* of the list.

The symbol table, $T$, the label table, $L$, and the function table, $F$, are content addressable memory. There is a hashing function, $hash(str, table)$, that takes as input a string representing a symbol, a label, or a function name, and a table and that adds $str$ to $table$ (if necessary) and returns the location in the table where the string is stored. $T$ stores the printable versions of symbols. Symbols can be added to the table using $addSymb(symb)$ which uses $hash(symb, T)$ and can be accessed using $fetchSymb(i)$ which returns, $T[i]$, the symbol in the $i^{th}$ position in $T$. $F$ stores *known functions* including primitives. $F[i]$, where $i = hash(f, F)$ for some function named $f$, stores a function record containing the string representing the name of the known function, $f$ ($F[i].name$), the address in code space for $f$ ($F[i].addr$) , the number of parameters $f$ has ($F[i].params$), and the length of $f$ ($F[i].len$ = the number of primitive instructions in the compiled code for $f$). $L[i]$, where $i = hash(l, L)$, for some label named $l$, stores a record containing the string representing a label ($L[i].name$) and the address in code space for $l$ ($L[i].addr$)

Each MT space can be thought of as a machine that provides services to and that hides the details of paging and the distributed virtual memory system from the evaluator. The evaluator requests a service from a given MT space and the given MT space provides the service. This is not suggesting that these are concurrent processes on the processor running the evaluator. It is simply an abstraction that will aid us with the discussion presented below. In the remainder of this section, we briefly describe the services provided by each MT space and results obtained from previous studies.

### 1.3.1 The MT Heap

The MT heap only stores dynamically allocated list-based structures. It provides two basic services to the evaluator: *heap allocation* and *heap access*. Heap cells are allocated linearly from beginning to end. The address of next heap cell available for allocation is kept in register *h*. The S-expression stored at address *i* is denoted by $H[i]$. The state of the heap can be described by a tuple $(H, h)$, where

$H = H[0], H[1], \ldots, H[h-1]$. A heap access, $hacc(i)$, does not change the state of the heap and returns $H[i]$. A heap allocation, $halloc(Sexp)$, changes the state of the heap. This behavior can be described by the transition rule:

$$(H, h) \rightarrow (H : H[h] = Sexp, h+1)$$

where $H : H[h] = Sexp$ represents the same heap as on the left of the arrow except that now $H[h] = Sexp$.

The MT heap allocation algorithm only allocates heap memory during the execution of **cons** and this allocation is delayed until both arguments to **cons** have been evaluated. Previous empirical studies have suggested that the MT allocation algorithm fosters locality of reference[MT00] and that $FIFO$ is as competitive as $LRU$ as a page replacement policy[MTN02a, MT03]. This property of being able to perform paging using $FIFO$ is very desirable, because $FIFO$ is easily implemented in software and does not incur a per access overhead like $LRU$. Keeping this property for the MT heap while supporting first-class functions is one of the motivating factors of the work presented in this article.

### 1.3.2 The MT Stack

Unlike the abstract machines described by Landin[P. 64], Henderson[Hen80], and Cardelli[Luc83, Car84], MT has only one stack. It used for parameter passing and flow control. For primitive functions, arguments are pushed onto the stack and the result is returned on top of the stack. For non-primitive known functions, an activation record, that consists of saved register values and the arguments to the function, is pushed onto the stack one element at a time. The first empty cell on the top of the stack is pointed to by register *s*. The S-expression stored at address *i* is denoted by $S[i]$. The state of the heap can be described by a tuple $(S, s)$, where $S = S[0], S[1], \ldots, S[s-1]$.

The MT stack, *S*, provides four services to the evaluator: *accessing the $i^{th}$ element on the stack* (stackAcc(i)), *pushing a value onto the stack* (push(Sexp)), *popping and returning the top element off the stack* (pop()), and *popping the n top elements off the stack* (npop(n)). These four services change the state of the stack and are described by the following transition rules:

$stackAcc(i)$: $(S, s) \rightarrow (S : S[s] = S[i], s+1)$
$push(Sexp)$: $(S, s) \rightarrow (S : S[s] = Sexp, s+1)$
$pop()$: $(S, s) \rightarrow (S, s-1)$[3]
$npop(n)$: $(S, s) \rightarrow (S, s-n)$.

where $S : S[s] = V$ represents the same stack at the left of the arrows except that now $S[s] = V$.

In previous studies on the MT stack, we have empirically validated that LRU is superior to FIFO as a page replacement policy and, in fact, have mathematically

[3]The value returned by pop() can be assigned to a register or temporary variable.

demonstrated that LRU is optimal for paging the MT stack[MTN02b]. The proof is based on the reasonable assumptions that *stackAcc*$(i)$ is only used to access values in the top activation record on the stack and that the size of every activation record is smaller than the size of a stack page. Based on this proof, the *MT stack page replacement algorithm* (MTSRA) was developed which implements LRU without the costly per access overhead associated with LRU. This property is very desirable for a software-based DVM like in MT, because there is no hardware support for paging. Maintaining a stack discipline that keeps LRU optimal for the MT stack is another motivating factor for the work presented here.

### 1.3.3 The MT Code Space

The MT code space, *C*, is used to store sequences of MTEVM primitives in sequential memory addreses. The cells of *C* are allocated linearly from beginning to end with PRIM tags. The next cell available for allocation is pointed to by the register *cp*. The services provided by the MT code space access and mutate *C*, *F*, *L*, and *cp*. $C[0..cp-1]$ is the complete set of primitive instruction sequences loaded into *C* and we denote the instruction at address *i* as $C[i]$. Services that do not alter the state of *C* include $fetchInstr(i)$ that returns $C[i]$, $fetchLabel[i]$ that returns the label record $L[i]$, and $fetchFunction[i]$ that returns the function record $F[i]$.

The services that change the state of code space are *add instruction* (addInstr(i)), *add label* (addLabel(l)), and *add function* (addFunct(fr)). The transitions describing their behavior are:

$addInstr(i)$: $(C,F,L,cp) \rightarrow (C : C[cp] = i, F, L, cp+1)$
$addFunct(fr)$: $(C,F,L,cp) \rightarrow (C, F : F[hash(fr.name, F)] = fr, L, cp)$
$addLabel(lr)$: $(C,F,L,cp) \rightarrow (C, F, L : L[hash(lr.name, L)] = lr, cp)$

where C:C[cp]=i, L:L[b]=lr, and F:F[a]=fr represent the same *C*, *L*, and *F* to the left of the arrows with $C[cp]$ changed to *i*, $L[b]$ changed to *lr*, and $F[a]$ changed to $fr$ respectively. *C* also provides the ability to load a file containing MT compiled code into *C* by looping through the file and applying $addInstr(i)$, $addFunct(fr)$, $addSymb(s)$, and $addLabel(lr)$.

## 1.4 THE MT VIRTUAL MACHINE EVALUATOR

MTEVM is a register-based machine that has access to the memory spaces of the MT System. These memory spaces and a set of registers define the state of MTEVM. A set of computation, flow control, and register-machine primitives are defined that operate on the MT spaces and the set of registers. In this section, we first informally define MTEVM and then present its operational semantics.

### 1.4.1 The Abstract Machine

The MTEVM implements a fetch-execute cycle that loops through the instruction stream stored in code space until it halts (i.e. reaches the end of the current

computation). At each iteration of the loop, an MTEVM primitive instruction is executed that changes the state of the machine. This set-up is a slight variation of abstract machines defined in, for example, [AwJS90, Hen80, P. 64] which store instruction sequences as lists. Nonetheless, the approaches are equivalent in that they define a mechanism to store the necessary information to execute the next instruction without necessarily defining this mechanism as a fetch-execute cycle.

The state of the MTEVM is determined by the heap (H), the stack (S), the code space (C), the symbol table (T), the function table (F), and 9 registers: the *program counter register* (p) that stores the virtual address of the next instruction in *C* to be executed, the *value register* (v) that is used to temporarily store the results of applications, the *environment register* (e) that stores the virtual address in *S* of the activation record of the current non-primitive known function being evaluated, the *continue register* (c) that stores the virtual address in *C* of the next instruction to be executed upon returning from a function call, the *new environment* (n) register that is used to store the virtual stack address of an activation record under construction for a non-primitive known function that is to be called, the *temporary register* (t) that is used as temporary storage for values of applications and to temporary store virtual *C* addresses when returning from a known function, the *stacktop regsiter* (s) that stores the virtual stack address of the next available cell at the top of *S*, the *next available heap address register* (h) that stores the virtual heap address of the next available cell for allocation *H*, and the *next available code space address register* (cp) that stores the virtual code space address of the next available cell for allocation in *C*.

Each MTEVM primitive mutates one or more of the memory spaces and/or registers. During the execution of a program stored in *C*, *T* is never changes. That is, the execution of compiled MT code does not add in or remove symbols from the symbol table. Changes to *T* can only occur when loading compiled code or when parsing user input.

### 1.4.2 Operational Semantics

The semantics of the MTEVM is given operationally by state transitions. The state of the machine can be denoted by a tuple representing each of the MT memory spaces and each of the registers: $(\text{p,v,e,c,n,t,s,h,cp,H,S,C,T,F})$ . We represent machine transitions by:

$$(\text{p,v,e,c,n,t,s,h,cp,H,S,C,T,L,F}) \rightarrow (\text{p',v',e',cp',n',t',s',h',c',H',S',C',T',F'}).$$

The tuple to the left of the arrow represents the state of the machine before executing the primitive referred to by *p* and the tuple to the right of the arrow represents the state of the machine after executing the primitive referred to by *p*. We write *x.S* for the stack to mean that the top value of the stack is *x*. Likewise, *x.y.S*, denotes that the top value of the stack is *x* and *y* is the next value on the stack. $C[p]$ denotes the next instruction to be executed. Finally, we denote the value of applying a primitive, $op$, as $(op\ arglist)$ where *arglist* are the arguments (stored on the top of the stack) to $op$ separated by blank spaces. For example, if

$C[p]$ is the primitive **not** then $(C[p]\ a)$ denotes (**not** $a$) for some MT S-expression on the top of the stack.

The MT primitives are divided into 4 categories: computation, flow control, register-machine, and I/O. Of interest to us here are the first three categories.

***Computation Primitives***

Computation primitives are those that compute a new value based on input on the top of the stack and push the resulting value onto the stack. We can further sub-categorize computation primitives by the number of inputs they expect on the top of the stack (1 or 2). For a one-input computation primitive, we have the following transition rule:

(p,v,e,c,n,t,s,h,cp,H,x.S,C,T,L,F) →
    (p+1,v,e,c,n,t,s,h,cp,H,(C[p] x).S,C,T,L,F)

For a two-input computation primitive, $C[p] \neq$**cons**, we have the following transition rule:

(p,v,e,c,n,t,s,h,cp,H,y.x.S,C,T,L,F) →
    (p+1,v,e,c,n,t,s-1,h,cp,H,(C[p] x y).S,C,T,L,F)

The above transition rule indicates that arguments to a primitive must be stored in reversed order on the stack. That is, (from left to right) the first argument must be pushed first and the second argument must be pushed second. This same convention for pushing arguments is used for non-primitive known functions.

The only computation primitive that changes $H$ is **cons**. Therefore, we must define its transition rule differently. For $C[p] =$**cons**, we have

(p,v,e,c,n,t,s,h,cp,H,y.x.S,C,T,L,F) →
    (p+1,v,e,c,n,t,s-1,h+2,cp,H:H[h]=x:H[h+1]=y,LIST:h:h+1.S,C,T,L,F)

The above transition rules states that the arguments to **cons**, $x$ and $y$, are allocated in heap positions $h$ and $h+1$ respectively, that the $h$ register is increased by 2 (for the two allocated values), the program counter is increased by 1, the stack top is decreased by 1, and a list S-expression with its $car = h$ and $cdr = h+1$ is pushed onto the stack.

***Flow Control Primitives***

Flow control primitives change the value of $p$ to continue execution at an instruction other than the next instruction at $p+1$ if the top of the stack contains anything other than a false value[4]. These primitives include conditional branches

[4]False values are defined as *nil* and *false* (#f)

and an unconditional branch. A conditional branching primitive is used to transfer control to a (compiler generated) label while unconditional branching is used to transfer control to a non-primitive known function. The conditional branching primitives have embedded within the instruction a label, *l*, that indicates the value *p* is to be updated to if the branch is taken. The unconditional branch primitive (GOTO) has embedded an index, *f*, into *F* where the value that *p* is to be updated to is stored.

The branch primitive, BR *l*, tests the value at the top of the stack. If that value is not a false value then the branch is taken and *p* is updated to *l*. Otherwise, *p* is incremented by 1. The state transitions for BR *l* are defined as follows:

For $x \neq false$ and $x \neq nil$:
(p,v,e,c,n,t,s,h,cp,H,x.S,C,T,L,F) → (l,v,e,c,n,t,s-1,h,cp,H,S,C,T,L,F).
For $x = false$ or $x = nil$,
(p,v,e,c,n,t,s,h,cp,H,x.S,C,T,L,F) → (p+1,v,e,c,n,t,s-1,h,cp,H,S,C,T,L,F).
The transition for GOTO *f* is:
$(p,v,e,c,n,t,s,h,cp,H,S,C,T,F) \rightarrow (F[f].addr,v,e,c,n,t,s,h,cp,H,S,C,T,L,F)$.

For GOTO, *p* is updated to the address in code space of the first instruction associated with *f*. This address is stored in F[f].

Conditional branching is optimized when the branching depends on the result of a computation primitive that returns a boolean. The transitions for these primitives are not shown.

***Register-Machine Primitives***

The register-machine primitives pop and push values from and to the stack, transfer values between registers, set the value of registers, perform parameter accesses, start building activation records, and return from non-primitive known functions.

There are primitives to save the value of a register onto the stack. They are represented by an opcode obtained from concatenating *s* to the *register name*. For example, *sv* saves the contents of the *v* register on the stack. The transitions for the *save register X* primitives, sX, are given by:

(p,v,e,c,n,t,s,h,cp,H,S,C,T,L,F) → (p+1,v,e,c,n,t,s+1,h,cp,H,X.S,C,T,L,F)

where *X*.*S* is *S* with the value of register *X* pushed onto it.

Conversely, we can restore the value of registers from the top of the stack. A restore primitive opcode is obtained in the same manner as a save primitive opcode except that it starts with *r* instead of *s*. The transition rule for $C[p] = rv$ (restore the *v* register) is:

$(p,v,e,c,n,t,s,h,cp,H,v'.S,C,T,L,F) \rightarrow (p+1,v',e,c,n,t,s+1,h,cp,H,S,C,T,L,F)$

The transition rules for restoring the other registers are analogous.

Constant values are pushed onto the stack by using the appropriate *mk**valuetype*** primitive. An integer, *i*, representing the constant being pushed is embedded with the instruction. For example, *MKSYM i* pushes a symbol S-expression onto the stack with *i*, the index into T of the symbol being pushed, as its value. The transition rule for *MKSYM i* is:

(p,v,e,c,n,t,s,h,cp,H,S,C,T,L,F) → (p,v,e,c,n,t,s+1,h,cp,H,SYMB:i.S,C,T,L,F)

where SYMB:i represents a symbol S-expression with value *i*. The rest of the *mk**valuetype*** primitives are analogous except for MKLIST that has no embedded input with the instruction and expects the index into the heap of the cdr to be the topmost value on the stack and the index into the heap of the car to be the second value from the top of the stack. The transition rule for MKLIST is:

(p,v,e,c,n,t,s,h,cp,H,j.i,S,C,T,L,F) → (p,v,e,c,n,t,s-1,h,cp,H,LIST:i:j.S,C,T,L,F)

The *POPN n* primitive decrements *s* by *n*. The transition rule is:

(p,v,e,c,n,t,s,h,cp,H,S,C,T,L,F) → (p,v,e,c,n,t,s-n,h,cp,H,S,C,T,L,F)

A stack value can be accessed and pushed onto the stack by using *PACC d*. The embedded value *d* is a displacement from the stack address stored in *e*. This primitive is used to access the parameters of a function using lexical addressing. The transition for $C[p] = PACC\ d$ is:

(p,v,e,c,n,t,s,h,cp,H,S,C,T,L,F) → (p+1,v,e,c,n,t,s,h,cp,H,S[e+d].S ,C,T,L,F)

There are two set primitives, $SETC\ i$ and $SETE\ i$, to set the contents of the *c* and *e* registers. Both primitives have an embedded input *i* which represents the virtual code space address of a label for $SETC\ i$ and the virtual stack address of an activation record for $SETE\ i$. Their transition rules are analogous. The transition rule for $SETC\ i$ is:

(p,v,e,c,n,t,s,h,cp,H,S,C,T,L,F) → (p+1,v,e,i,n,t,s,h,cp,H,S,C,T,L,F)

Register to register transfer operations are represented by an opcode starting with a register name followed by '2' and ending with a register name. For example, the primitive *c2p* transfers the contents of register *c* to register *p*. The transition rules for these primitives are all analogous. The transition rule for *c2p* is:

(p,v,e,c,n,t,s,h,cp,H,S,C,T,L,F) → (c,v,e,c,n,t,s,h,cp,H,S,C,T,L,F)

Finally, there are primitives to call and return from known functions that are not computation primitives. *FCALL* starts building an activation record by saving

the the *e* and *c* registers on the stack. *FRETURN n* returns from a call to a function with n arguments by saving its result, $(f\ x_1 \ldots x_n)$, in *v*, popping the n arguments off the stack, setting *p* to *c*, restoring *c* and *e*, and pushing $(f\ x_1 \ldots x_n)$ back onto the stack. The transition rule for *FCALL* is:

$$(p,v,e,c,n,t,s,h,cp,H,S,C,T,L,F) \rightarrow (p{+}1,v,e,c,n,t,s{+}2,h,cp,H,c.e.S,C,T,L,F)$$

and for *FRETURN* the transition rule is:

$$(p,v,e,c,n,t,s,h,cp,H,(f\ x_1 \ldots x_n).x_1 \ldots x_n.c'.e'.S,C,T,L,F) \rightarrow$$
$$(c,(f\ x_1 \ldots x_n),e',c',n,t,s{-}n{-}2,h,cp,H,(f\ x_1 \ldots x_n).S,C,T,L,F)$$

### 1.5 FIRST-CLASS FUNCTIONS IN MT

The MTEVM as defined in the previous section can be used to implement a pure list-based functional language that does not have first-class functions. In this section, we outline the support provided for first-class functions. One of the goals is to prevent the the mixing of data lists with closures in the heap, because it makes it difficult to predict how heap pages will be accessed. For example, the same data list, *L*, can be passed in as the argument to many different functions (that traverse *L*) represented by heap-allocated closures. In MT, we expect $FIFO$ to perform as well as *LRU* for managing heap pages when *L* is traversed. This, however,can not be expected if closures are heap-allocated.

In functional languages with heap-alloacted closures, determining the value a free variable, $fv$, requires that a closure be accessed[Hen80, Luc83, Car84]. This is inefficient when compared to accessing a parameter in an activation record on the stack or an embedded value within an instruction. When a closure is created, the bindings of the free variables in the function, $f$, it represents are known. Given that these bindings are known, it is desirable to optimize the accessing of these variables to be as fast as possible when $f$ is applied.

In the context of a pure functional language, we can use partial evaluation to prevent the mixing of closures and list data in the heap and to make accesses to free variables faster. We can observe that when a closure is created in a pure functional language, the bindings of the free variables remain unchanged throughout the existence of $f$. Therefore, instead of heap allocating a closure for $f$ it is safe to create a new function where there are no accesses to free variables. In the remainder of this section, the operational semantics for MTEVM primitives used to implement first-class functions is given.

#### 1.5.1 Passing, Returning, and Calling Functions

To pass in a known function as an argument to a function or to return a known function as the value of a function, we need a primitive $MKFUNCT\ f$ (analogous, for example, to $MKINT\ i$) whose embedded argument $f$ is an index into $F$. If $f$

is not the index of a primitive operation then an S-expression with a *FUNCT* tag and a value of $f$ is pushed onto the stack. The transition rule is:

(p,v,e,c,n,t,s,h,cp,H,S,C,T,L,F) →
(p+1,v,e,c,n,t,s+1,h,cp,H,FUNCT:f.S,C,T,L,F).

If $f$ indexes a computation primitive, represented by *prim*, then the transition rule is:

(p,v,e,c,n,t,s,h,cp,H,S,C,T,L,F) →
(p+1,v,e,c,n,t,s+1,h,cp,H,PRIM:prim.S,C,T,L,F).

Consider the Scheme function: (define (h op x) (op 5 x)). We can not use the *GOTO* primitive to call a function that has been been passed in as an argument to a function (or has been returned as a value from a function), because *op* may be a primitive. Therefore, we define a new *GOTO* primitive, *GOTOVF*, that looks at the top of the stack, to determine the next state of the machine. The call to *op* in the body of *h* is implemented by executing *PACC* 1 *GOTOVF* which pushes onto the stack the S-expression representing *op* before executing *GOTOVF*. The transition rules for *GOTOVF* are:

(p,v,e,c,n,t,s,h,cp,H,FUNCT:f.S,C,T,L,F) →
(F[f].addr,v,e,c,n,t,s-1,h,cp,H,S,C,T,L,F)

(p,v,e,c,n,t,s,h,cp,H,PRIM:prim.a.S,C,T,L,F) →
(p+1,v,e,c,n,t,s-1,h,cp,H,(prim a).S,C,T,L,F)

(p,v,e,c,n,t,s,h,cp,H,PRIM:prim.a.b.S,C,T,L,F) →
(p+1,v,e,c,n,t,s-2,h,cp,H,(prim a b).S,C,T,L,F) .

Two transition rules are needed for an S-expression with a *PRIMTAG*, because a the primitive may take 1 or 2 arguments which can be determined by examining its value *prim*. We note that before pushing the arguments the function will be applied to it is necessary to examine the function to determine if it is not a primitive. If so, *FCALL* must be used to start building its activation record.

### 1.5.2 Anonymous Functions

Anonymous functions are functions that lack a name. In Scheme, the special form **lambda** is used to create an anonymous function. For example, the lambda function (lambda (x) (* x x)) creates an anonymous function that that squares its input. To support anonymous functions a heap-allocated closure could be created, but partial evaluation can be used to treat anonymous functions as known functions at runtime. To do so in Scheme, before compilation every lambda expression can be lifted and defined as a known function that returns a function. We will distinguish

two cases for matters of efficiency at runtime: lambda expressions lacking free variables and those containing free variables.

Consider the expression:

$$((lambda\ (x)\ (*\ x\ x))\ 5)$$

The lambda expression without free variables and can be lifted to create:

$$(define\ (UniqueFName\ x)\ (*\ x\ x)).$$

The lifting of a lambda expression with no free variables creates a function whose parameters are the parameters of the original lambda expression. We can now transform the original expression to:

$$(UniqueFName\ 5)$$

*UniqueFName* is a known function and the evaluation of the above expression is described with the primitives from the previous section. Furthermore, *UniqueFName* can be created at compile time forcing no overhead in the evaluation of the lambda expression (that exists at the Scheme level) at runtime and can be loaded into $C$ and added to $F$ before any execution takes place.

Supporting anonymous functions that contain free variables is more complex, because we are unable to create a known function for them at compile time. When a lambda expression with free variables, *alam*, is evaluated, however, we can create a known function for it, load it to $C$, and add it to $F$ by using partial evaluation. To support this, we define a new primitive to make a known function at runtime, $MKKFUNCT\ l$, that expects on top of the stack the bindings of the free variables in *alam* and that returns on the top of the stack an S-expression for a new known function, *UAF*, that contains no accesses to free variables. The embedded argument $l$ is an index into $C$ where a compiler generated function to create specialized functions for *alam*, *gac*, is stored. That is, $l$ points to a code generator tailored-made for *alam*. This idea is similar to the approach used to optimize ML with runtime code generation[Pet96, Mar94]. $MKKFUNCT\ l$ will temporarily transfer control to $l$ (which changes $F$ and $C$, and may change $H$ and $L$), pop the bindings of the free variables of *alam*, and push the S-expression for *UAF* onto the stack. The transition rule for $MKKFUNCT\ l$ is:

$(p,v,e,c,n,t,s,h,cp,H,x_n \ldots x_1.S,C,T,F) \rightarrow$

$(p+1,v,e,c,n,t,s\text{-}n+1,h,cp+len(UAF),H',\text{FUNCT:hash(UAF,F).S},C',T,L',F')$

where $x_n \ldots x_1$ are the bindings of the free variables of *alam*, $len(UAF) =$ the length of the compiled code of function *UAF*, $C' = C : C[\text{cp}]..C[cp+ \text{ len(UAF) -1}] = UAF$, $F' = F : F[hash(UAF,F)] =$ the function record for UAF, $L' = L$ with the label records for UAF added, and $H'$ is $H$ with any allocations done by *gac* for *UAF*.

*UAF* is the code obtained from applying partial evaluation to to *alam* and its free variables. The result of this application can be obtained by simply substituting accesses to free variables in $C[a]..C[b]$ with the appropriate MTEVM primitive to push a constant onto the stack, by reducing to normal form subexpressions in *alam* that depend only on its free variables (any lists computed would account for changes in $H'$), and/or by recursively applying partial evaluation to subexpressions in *alam* that do not solely depend on its free variables. Certainly, the value of computation primitives that depend only on free variables should be computed when *UAF* is created. The extent to which further partial evaluation should be applied is to be determined empirically by testing different heuristics. For example, it may not be worth the investment to evaluate or apply partial evaluation to both the ***then*** and ***else*** clauses of an ***if*** statement.

To illustrate how support for anonymous functions works, consider the following Scheme definition:

$$(define\ (myf\ x\ L)\ (map\ (lambda\ (y)\ (+\ x\ y))\ L))$$

and the call $(myf\ 10\ '(1\ 2 \ldots 1000000))$. The specialized code for the lambda expression looks like:

$UAFmyf_{x=10}$:
```
MKINT 10
PACC 1
ADD
FRETURN
```

The specialized function, $UAFmyf_{x=10}$, is a known function and may be applied to each element of the list as if it had been defined by a programmer. To access the value of the free variable *x*, there is no searching involved and it is achieved by executing one primitive instruction ($MKINT\ 10$) to push a constant onto the stack. This is at least as fast as accessing the value of the bound variable *y* ($PACC\ 1$).

## 1.6 CONCLUDING REMARKS

The MT system is being developed to understand and improve the interaction between a pure list-based functional language and memory. To this end, MT divides memory into different spaces in order to improve their management. One of the goals is to develop a system of memory allocation from which we can derive reasonable expectations on how memory space is accessed at runtime. In this article, a brief description of the semantics of the MT Evaluator Virtual Machine and a description of the support provided for first-class functions has been presented. Support for first-class functions is provided without allocating MT-heap memory and without changing the expected access patterns of the MT heap and MT stack. This means that $FIFO$ can continue to be used for the MT heap and $MTSRA$ (our implementation of LRU) for the MT stack rendering first-class functions in MT

DVM friendly. First-class functions are supported by using partial evaluation and allocating memory for them in the MT code space.

The work presented suggests several interesting lines of research that can be pursued. It is important to determine how code space pages should be swapped between the evaluator and the backing store DVM system. It is unclear, for example, if branching or support for first-class functions will render either $FIFO$ or *LRU* ineffective. Now that code space is not static, questions on how and where allocations should take place arise. Should known functions to a programmer be kept separate from known functions created for anonymous functions? The dynamic nature of code space also raises the question of how should code space be garbage collected and/or compacted. Finally, the effectiveness of partial evaluation will depend on how much partial evaluation can be performed without making program evaluation slower. We may discover, for example, that for an eager language (like Scheme) a lazy partial evaluator works best. The avenues for research are, indeed, exciting.

## 1.7 ACKNOWLEDGEMENTS

The author would like to thank Barbara Mucha, Victor Encarnacion, and Kristine Apon for their past and continued efforts in making MT an implemented reality. Thanks are also in order for Greg Michaelson who provided insightful comments on some of the preliminary ideas presented in this article. Finally, the support of my department and the Seton Hall University Research Council have made this work possible.

## REFERENCES

[AM91] Andrew W. Appel and David B. MacQueen. Standard ML of New Jersey. In *Proceedings of The Third International Symposium on Programming Language Implementation and Logic Programming*, pages 1–13. Springer-Verlag, Aug 1991.

[AS96] Andrew W. Appel and Zhong Shao. An Empirical and Analytical Study of Stack vs. Heap Cost for Languages with Closures. *Journal of Functional Programming*, 6(1):47–74, 1996.

[AwJS90] H. Abelson and J. Sussman with Julie Sussman. *Structure and Interpretation of Computer Programs*. McGraw-Hill, 1990.

[BF00] Clem Baker-Finch. An Abstract Machine for Parallel Lazy Evaluation. In Michaelson et al. [MTL00], pages 153–161.

[Car84] Luca Cardelli. Compiling a Functional Language. In *Proceedings of the 1984 ACM Conference on LISP and Functional Programming*, pages 208–217, New York, 1984. ACM Press.

[Chr90] Chris Hanson. Efficient Stack Allocation for Tail-Recursive Languages. In *ACM Conference on Lisp and Functional Programming*, pages 106–118, New York, June 1990. ACM Press.

[Chu41] Alonzo Church. *The Calculi of Lambda-Conversion*. Number 6 in Annals of Mathematics Studies. Princeton University Press, Princeton, NJ, 1941.

[Dij62] Edsger W. Dijkstra. Substitution processes. Available from http://www.cs.utexas.edu/users/EWD/ewd00xx/EWD28.PDF, January 1962.

[Hen80] Peter Henderson. *Functional Programming: Application and Implementation*. Prentice-Hall International, Englewood, NJ, USA, 1980.

[Joh02] John Weber. Scheme48 Virtual Machine Developer's Guide. Available from the author, July 2002.

[Jon96] Neil D. Jones. An Introduction to Partial Evaluation. *ACM Computing Surveys*, 28(3):480–503, 1996.

[Ken97] Kenechi Asai, Hidehiko Masuhara and Akinori Yonezawa. Partial Evaluation of Call-by-Value $\lambda-$calculus with Side-effects. In *Proceedings of the ACM SIGPLAN Symposium on Partial Evaluation and Semantics-Based Program Manipulation*, volume 32 of *Sigplan Notices*, pages 12–21, New York, 1997. ACM Press.

[Loi01] Hans-Wolfgang Loidl. Load Balancing in a Parallel Graph Reducer. In Kevin Hammond and Sharon Curtis, editors, *Trends in Functional Programming*, volume 3, pages 63–74, Bristol, UK, 2001. Intellect. ISBN 1-8410-070-4.

[Luc83] Luca Cardelli. The Functional Abstract Machine. Technical Report No.107, Bell Laboratories, April 1983.

[LY96] T. Lindholm and F. Yellin. *The Java Virtual Machine Specification*. Addison-Wesley, Reading, MA, 1996.

[Mad98] Mads Tofte. A Brief Introduction to Regions. In *Proceedings of the 1998 ACM International Symposium on Memory Management*, pages 186–195, 1998.

[Mar94] Mark Leone and Peter Lee. Lightweight Run-Time Code Generation. In *Proceedings of the ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, pages 97–106. Technical Report 94/9, Department of Computer Science, University of Melbourne, June 1994.

[Moo84] David A. Moon. Garbage Collection in a Large Lisp System. *Proc. of the 1984 ACM Symp. on Lisp and Functional Programming*, pages 235–246, 1984.

[MT00] Marco T. Morazán and Douglas R. Troeger. The MT Architecture and Allocation Algorithm. In Michaelson et al. [MTL00], pages 97–104.

[MT03] Marco T. Morazán and Douglas R. Troeger. List-Heap Paging in a Distributed Virtual Memory System for Functional Languages. In Hamid Arabnia, editor, *Proceedings of the International Conference on Parallel and Distributed Processing Techniques and Applications*, pages 1689–1695. CSREA Press, 2003.

[MTL00] Greg Michaelson, Phil Trinder, and Hans-Wolfgang Loidl, editors. *Trends in Functional Programming*, volume 1, Bristol, UK, 2000. Intellect.

[MTN02a] Marco T. Morazán, Douglas R. Troeger, and Myles Nash. Paging in a Distributed Virtual Memory. In Kevin Hammond and Sharon Curtis, editors, *Trends in Functional Programming*, volume 3, pages 75–86, Bristol, UK, 2002. Intellect.

[MTN02b] Marco T. Morazán, Douglas R. Troeger, and Myles Nash. The MT Stack: Paging Algorithm and Performance in a Distributed Virtual Memory System. *CLEI Electronic Journal*, 5(1), 2002.

[P. 64] P. J. Landin. The Mechanical Evaluation of Expressions. *The Computer Journal*, 6(4):308–320, 1964.

[Pet96] Peter Lee and Mark Leone. Optimizing ML with Run-Time Code Generation. In *Proceedings of the ACM SIGPLAN Conference on Programming Language Design and Implementation*, pages 137–148. ACM Press, May 1996.

[S. 98] S. Breitinger and U. Klusik and R. Loogen and Y. Ortega-Mallón and R. Peña. DREAM: the Distributed Eden Abstract Machine. In Chris Clack, Kevin Hammond, and Antony J. T. Davie, editors, *IFL*, volume 1467 of *Lecture Notes in Computer Science*, pages 250–269. Springer-Verlag, 1998.

[SA94] Zhong Shao and Adrew W. Appel. Space Efficient Closure Representations. In *Proceedings of the 1994 ACM Conference on LISP and Functional Programming*, pages 150–161, New York, 1994. ACM Press. ISBN 0-89791-643-3.

[Ste00] Stephan Diehl and Pieter Hartel and Peter Sestoft. Abstract Machines for Programming Language Implementation. *Future Generation Computer Systems*, 16(7):739–751, May 2000.

[Wil92] Paul R. Wilson. Uniprocessor Garbage Collection Techniques. In Yves Bekkers and Jacques Cohen, editors, *Proceedings of the 1992 International Workshop on Memory Management*, number 637 in Lecture Notes in Computer Science, pages 1–42, Saint-Malo, France, 1992. Springer-Verlag.